Apunte 21 - Spring Security

Introducción

Como se mencionó en el apunte anterior, la seguridad de una aplicación es un tema muy amplio, del que apenas se puede rascar la superficie en esta serie de apuntes. Considerando esta realidad, en las siguientes secciones se encontrarán una introducción a Spring Security y los detalles de implementación necesarios para agregar autorización en las llamadas a la API de un backend.

Spring Security

Como se estudió anteriormente, en el núcleo del framework Spring se encuentra un contenedor Inversión de Control e Inyección de Dependencias, sin embargo, Spring ha tenido un crecimiento muy grande y no se compone únicamente por este núcleo sino que consiste en una cantidad (francamente impresionante) de componentes y proyectos que trabajan en conjunto con el mismo para facilitar el acceso a datos, el desarrollo de endpoints, integraciones, seguridad, etc.. A esto, además, se le debe sumar la introducción de SpringBoot que permite reducir mucho el código necesario para iniciar el desarrollo de una aplicación, proveyendo una auto-configuración que puede ser personalizada de ser necesario.

Spring Security es uno de los proyectos de Spring que consiste en un framework de *Autenticación* y *Control de acceso*, dos temáticas que se han abordado en el apunte *Seguridad en el Backend*, y que ahora serán puestas en práctica haciendo uso de este proyecto como dependencia.

Objetivos

Tal como se indicara en el apartado de *Introducción*, el objetivo con el que se utilizará a *Spring Security* es el de proteger el acceso al API de un Backend de accesos no autorizados.

Incorporación de Spring Security a un proyecto SpringBoot

Utilizando SpringBoot es verdaderamente fácil incorporar nuevos componentes a un proyecto; basta agregar la dependencia necesaria en el archivo **pom.xml** del proyecto (si se está utilizando Maven). En caso de Utilizar *Spring Initializer*, es suficiente con solicitar la dependencia *Spring Security*.

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
```

Al agregar esta dependencia y ejecutar el proyecto, se podrá ver una salida similar a la siguiente:

```
Using generated security password: 99c85142-719a-42f3-a78e-853d1bc93a6b

This generated password is for development use only. Your security
configuration must be updated before running your application in
production.
```

Esto indica que se está utilizando un password generado automáticamente pero, ¿un password para qué?. Siendo que SpringBoot sigue la filosofía *Convention over configuration*, al agregarse *Spring Security*, su configuración por defecto indica que se deben proteger los endpoints. Entonces, si se quisiera acceder a un endpoint sin proveer un usuario y un password, la petición fallaría con un código de respuesta 401 Unauthorized, como se puede ver a continuación:

```
HTTP/1.1 401
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Connection: keep-alive
Content-Length: 0
Date: Sat, 14 Oct 2023 23:01:08 GMT
Expires: 0
Keep-Alive: timeout=60
Pragma: no-cache
Set-Cookie: JSESSIONID=6076B8D80729FCE4DDD9B23DA2B2A3B3; Path=/; HttpOnly
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
```

De la repuesta anterior, resaltan dos elementos: el primero es el código de respuesta 401 (como se esperaba) y el segundo es el header *WWW-Authenticate: Basic realm="Realm"* que indica que la autenticación esperada es mediante el esquema **HTTP Basic** que se describió en el *Apunte 16*.

Si se quiere acceder, entonces, a un endpoint deberá hacerse mediante una autenticación **HTTP Basic** con el usuario **user** y el password que mostró por consola el proyecto al iniciar (en este ejemplo: **99c85142-719a-42f3-a78e-853d1bc93a6b**).

Pero claro, es posible que para el Backend que se quiera desarrollar, este tipo de autenticación no sea la adecuada. Es posible que se quisieran utilizar OTPs, o una solución basada en tokens o, incluso, que se quiera delegar el proceso de autenticación/autorización a un componente que no forma parte del Backend, como podría ser un servidor de autenticación. En este escenario será necesario instruir a *Spring Security* al respecto y realizar las configuraciones necesarias para involucrar a todos los actores del caso.

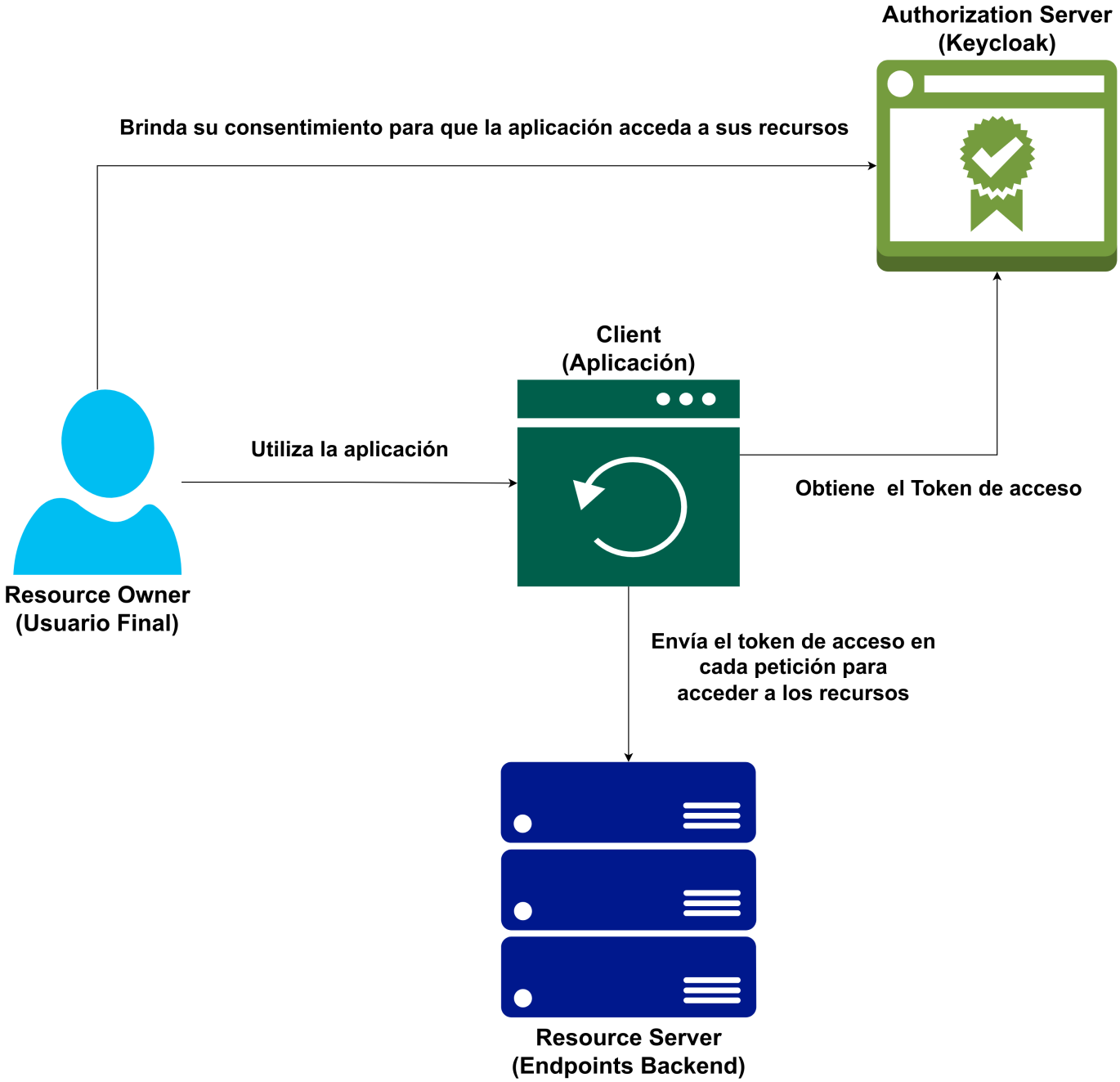
Configuración de Spring Security para implementar un Resource Server

Spring Security soporta escenarios variados, incluyendo la autenticación básica con usuario y password, además de toda la infraestructura para almacenar de forma segura al conjunto de usuarios y passwords permitidos en el Backend; incluyendo, también, la posibilidad de trabajar con tokens e, incluso, la delegación de autenticación/autorización con frameworks como *OAuth2* u *OIDC* siendo, este último, el enfoque que se adoptará en este apunte.

En el apunte anterior se explicaron los elementos fundamentales de los frameworks *OAuth2* y *OpenID Connect*, indicando que los mismos resolvían el problema de la delegación de acceso a terceras partes (básicamente a la aplicación que quería acceder al backend), y que esta solución se basaba en el uso de Tokens; esto significa que la aplicación, luego de que el usuario final diera su consentimiento, conseguía un token que es el que utilizaba para acceder a los recursos del backend. También se indicó que ambos definían una serie de actores:

- Un usuario final (*Resource Owner* para *OAuth2* o *End User* Para *OIDC*)
- La aplicación que quiere acceder al backend en nombre del usuario final (*Client* para *OAuth2*, *Relying Party* para *OIDC*)
- La API de Backend, el Servidor con los recursos a proteger (*Resource Server*)
- El Servidor de Autenticación/Autorización (*Authorization Server* para *OAuth2*, *OpenID Provider* para *OIDC*).

Se puede ver en la siguiente figura la operatoria general de estos protocolos:



Esta sección está dedicada a configurar únicamente el *Resource Server* que, en este contexto, no debe permitir invocaciones a sus APIs, si un usuario no está autenticado de acuerdo a los requerimientos del mismo.

Para el ejemplo de esta sección, el *Resource Server* contará únicamente con tres endpoints, que simplemente devolverán un mensaje. Además, para este ejemplo, se definirán dos roles *USUARIO* (usuarios no administradores) y *ADMIN* (administradores). Los requisitos para los mismos son:

- GET ***/publico***: Es un endpoint de acceso no controlado, no es necesario estar autenticado para consultarlo
- GET ***/protegido-usuarios***: Para invocarlo se requiere un usuario autenticado y con rol "*USUARIO*" o rol "*ADMIN*"
- GET ***/protegido-administradores***: Para invocarlo se requiere un usuario autenticado y con rol "*ADMIN*"

Será necesario, entonces, realizar las configuraciones necesarias para indicarle a *Spring Security* que debe permitir el acceso a estos endpoints, únicamente si se cumplen estos requisitos.

En esta sección se está desarrollando únicamente el *Resource Server*, que debe validar los tokens que se le presentan y extraer información de los mismos (por ejemplo, los roles que tiene el usuario o la identidad del mismo), para esto necesitará alguna comunicación con *Keycloak* (el *Authorization Server*) para poder validar que fue este último el que emitió el token.

Dependencias

Se indicó en la sección anterior que se desea programar un *Resource Server* donde las peticiones estén autorizadas. Para esto, en un proyecto SpringBoot se requerirán las siguientes dependencias:

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-resource-
server</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-oauth2-jose</artifactId>
        <version>6.1.4</version>
    </dependency>
```

Configuraciones

En principio la configuración básica consiste en indicarle a *Spring Security* cuál es la ubicación del servidor que emite los tokens (*Keycloack* en este caso), para que pueda comunicarse con él para conseguir los elementos que le permitirán comprobar que un token es válido.

En este documento, y para los ejemplos, se hará uso del instructivo de despliegue y configuración de Keycloak, considerando los siguientes parámetros:

- El realm se llama: **bda-realm**
- Existen dos roles: **ADMIN** y **USUARIO**
- Hay dos usuarios:
  - **usuario1**, password: **usuario123**
  - **admin1**, password: **admin123**
- Hay un cliente: **bda-client** sin client secret
- El servidor se desplegó en http://localhost:8081

Esta configuración se agrega en el archivo *application.yml* (o *application.properties*) de la siguiente manera:

Archivo *application.yml*:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/bda-realm
```

Archivo *application.properties*:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri =
http://localhost:8081/realms/bda-realm
```

Simplemente con esta configuración, los distintos endpoints requeriran un token de acceso válido para poder ser invocados. Hay que tener en cuenta lo siguiente:

- En la configuración por defecto todos los endpoints quedarán protegidos (no se podrá seleccionar cuáles requieren autorización y cuáles no)
- Con esta configuración se está indicando que los tokens serán del tipo JWT

Personalización

Una manera de sobrescribir la configuración por defecto de un módulo de *SpringBoot* es mediante clases anotadas con *@Configuration* y proveer una versión propia del Bean (o los Beans) que configuran al módulo. La siguiente configuración muestra un ejemplo de cómo se podría seleccionar cuáles son las rutas protegidas:

```
@Configuration
@EnableWebSecurity
public class ResourceServerConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
                http.authorizeHttpRequests(authorize -> authorize
```

```
                // Esta ruta puede ser accedida por cualquiera, sin
autorización
                .requestMatchers("/publico/**")
                .permitAll()

                // Esta ruta puede ser accedida únicamente por usuarios
autenticados
                .requestMatchers("/protegido-administradores/**")
                .authenticated()

                // esta ruta puede ser accedida únicamente por usuarios
autenticados
                .requestMatchers("/protegido-usuarios/**")
                .authenticated()

        ).oauth2ResourceServer(oauth2 ->
oauth2.jwt(Customizer.withDefaults()));
        return http.build();
    }
}
```

Esta configuración cubre una parte de los requerimientos planteados para los endpoints (acceso libre a */publico* y acceso protegido a */protegido-administradores* y */protegido-usuarios*), pero todavía no tiene en cuenta los roles. Esto significa que cualquier token válido permitirá acceso a los recursos protegidos, sin importar el rol que tenga el usuario. Para explicitar qué roles pueden acceder a cada endpoint, la configuración podría ser la siguiente:

```
@Configuration
@EnableWebSecurity
public class ResourceServerConfig {
    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(authorize -> authorize
                // Esta ruta puede ser accedida por cualquiera, sin
autorización
                .requestMatchers("/publico/**")
                .permitAll()

                // Esta ruta puede ser accedida únicamente por usuarios
autenticados con el rol
                // de administrador
                .requestMatchers("/protegido-administradores/**")
                .hasRole("ADMIN")

                // esta ruta puede ser accedida únicamente por usuario
autenticados
                // con el rol de usuario o administrador
                .requestMatchers("/protegido-usuarios/**")
                .hasAnyRole("USUARIO", "ADMIN")
```

```
                // Protegido mixto, GET para "USUARIO" y "ADMIN"
                .requestMatchers(HttpMethod.GET, "/protegido-mixto/**")
                .hasAnyRole("USUARIO", "ADMIN")

                // Protegido mixto, POST para "ADMIN"
                .requestMatchers(HttpMethod.POST, "/protegido-mixto/**")
                .hasRole("ADMIN")

                // Cualquier otra petición...
                .anyRequest()
                .authenticated()

        ).oauth2ResourceServer(
                oauth2 -> oauth2.jwt(jwt ->
jwt.jwtAuthenticationConverter(jwtAuthenticationConverter())));
        return http.build();
    }

    @Bean
    Converter<Jwt, AbstractAuthenticationToken>
jwtAuthenticationConverter() {
        return new Converter<Jwt, AbstractAuthenticationToken>() {
            @Override
            public AbstractAuthenticationToken convert(Jwt jwt) {
                // Solicito el claim con el que keycloak manda el mapeo de
roles
                // (contiene un objeto llamado "roles" cuyo contenido es
una lista de String)
                Map<String, List<String>> realmAccess =
jwt.getClaim("realm_access");

                // Se toma cada rol de la lista y se lo convierte a un
objeto GrantedAuthority
                List<GrantedAuthority> authorities =
realmAccess.get("roles")
                        .stream()
                        .map(r ->
String.format("ROLE_%s",r.toUpperCase())) // Spring security espera este
formato
                        .map(SimpleGrantedAuthority::new)
                        .collect(Collectors.toList());

                // Devuelvo el objeto de autenticación
                return new JwtAuthenticationToken(jwt, authorities);
            }
        };
    }


}
```

Con esta configuración, se puede cumplir con los requisitos indicados previamente pero, claramente, *Spring Security* brinda muchas más opciones de configuración de la autenticación y, además, esto se puede aplicar en distintas capas y no únicamente en los controladores.

A modo de ejemplo, se podría agregar un nuevo endpoint: "***/protegido-mixto***" donde GET puede ser invocado con el rol *USUARIO* o *ADMIN* pero POST únicamente por *ADMIN*. Esto se puede configurar de la siguiente manera:

```
@Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests(authorize -> authorize
                // Esta ruta puede ser accedida por cualquiera, sin
autorización
                .requestMatchers("/publico/**")
                .permitAll()

                // Esta ruta puede ser accedida únicamente por usuarios
autenticados con el rol de administrador
                .requestMatchers("/protegido-administradores/**")
                .hasRole("ADMIN")

                // esta ruta puede ser accedida únicamente por usuario
autenticados
                // con el rol de usuario o administrador
                .requestMatchers("/protegido-usuarios/**")
                .hasAnyRole("USUARIO", "ADMIN")

                // Protegido mixto, GET para "USUARIO" y "ADMIN"
                .requestMatchers(HttpMethod.GET, "/protegido-mixto/**")
                .hasAnyRole("USUARIO", "ADMIN")

                // Protegido mixto, POST para "ADMIN"
                .requestMatchers(HttpMethod.POST, "/protegido-mixto/**")
                .hasRole("ADMIN")

                // Cualquier otra petición...
                .anyRequest()
                .authenticated()

        ).oauth2ResourceServer(oauth2 ->
oauth2.jwt(Customizer.withDefaults()));
        return http.build();
    }
```

Y, aunque esta configuración podría ser suficiente en algunos escenarios, igualmente existe más flexibilidad para realizar comprobaciones de autorización. Una opción adicional consiste en habilitar este chequeo a nivel de métodos. Esto se realiza con la anotación **@EnableMethodSecurity** en la configuración, como se muestra a continuación:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        ...
    }
    ....

}
```

Con esta nueva configuración, se pueden agregar comprobaciones, por ejemplo, en el método de un servicio, de la siguiente manera:

```
@Service
public class RecursoProtegidoServiceImpl
        implements RecursoProtegidoService{

    @Override
    @PreAuthorize("#idUsuario == authentication.name")
    public String buscarPorIdUsuario(String idUsuario) {
        return "Este es un recurso protegido, solamente accesible por su
dueño!";
    }
}
```

De esta manera se puede solicitar a *Spring Security* que antes de ejecutar el método *buscarPorIdUsuario*, verifique que el parámetro que se le pasa, coincida con el nombre asociado a la autenticación. Cabe aclarar que, por defecto, *authentication.name* hace referencia al claim *sub* cuando se trabaja con JWT (que, generalmente contiene el id del usuario), pero esto puede ser modificado para que responda a algún otro claim.

Accediendo al objeto de Autorización

En la sección anterior, al habilitar la seguridad por métodos, se hizo referencia al objeto "authorization" pero, ¿se puede acceder a este objeto?. La repuesta es sí, y se puede hacer de varias maneras. Una de ellas es declararlo como un parámetro de los métodos de los controladores:

```
    @GetMapping("/{id}")
    public ResponseEntity<String> getRecurso(@AuthenticationPrincipal Jwt
jwt, @PathVariable("id") String id) {
        log.debug("Auth Name: {}", jwt.getSubject());
```

```
        return ResponseEntity.ok(service.buscarPorIdUsuario(id));
    }
```

Otra manera es acceder al *SecurityContext* de la siguiente manera:

```
Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
```

Obtención de un Token

Esta sección ya se encuentra detallada en el instructivo de despliegue y configuración de KeyCloak, sin embargo aquí se detalla de qué manera se podría hacer lo mismo utilizando Postman. Se debe recordar que el foco del presente apunte es la verificación del token, y no su obtención.

En las secciones anteriores el foco estuvo puesto en el *Resource Server*, básicamente se analizó de qué manera este componente puede proteger a los recursos, solicitando la autorización necesaria para que se pueda invocar a los distintos endpoints. Este componente es la esencia de la materia Backend de Aplicaciones, y por eso que la atención está puesta allí.

Sin embargo, para poder probar los endpoints del backend es necesario obtener un token para agregarlo a cada invocación.

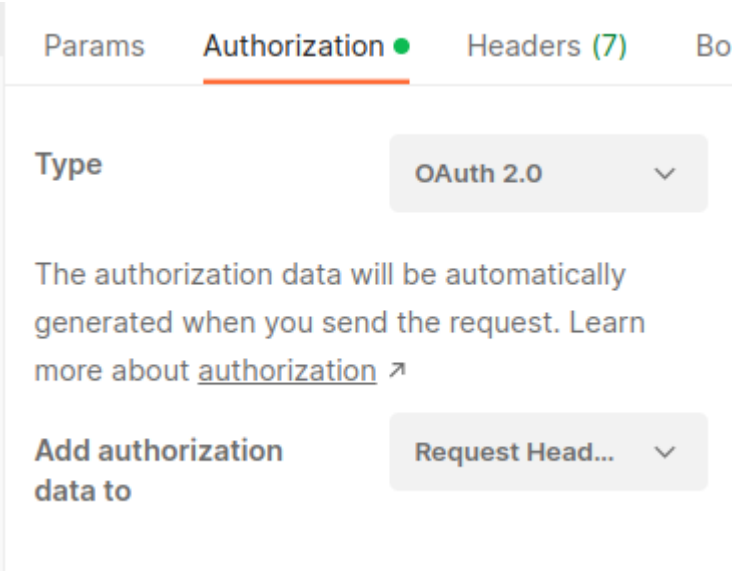
Como se discutió en el apunte anterior, existen distintos "grant types" o "flujos" para conseguir esto, que dependen de las características del cliente, entre otras cosas. Afortunadamente, se puede utilizar Postman para que obtenga el token.

Nota: Para los siguientes pasos, se asumen las mismas configuraciones mencionadas previamente, respecto a la URI de Keycloak, usuarios, roles, etc.

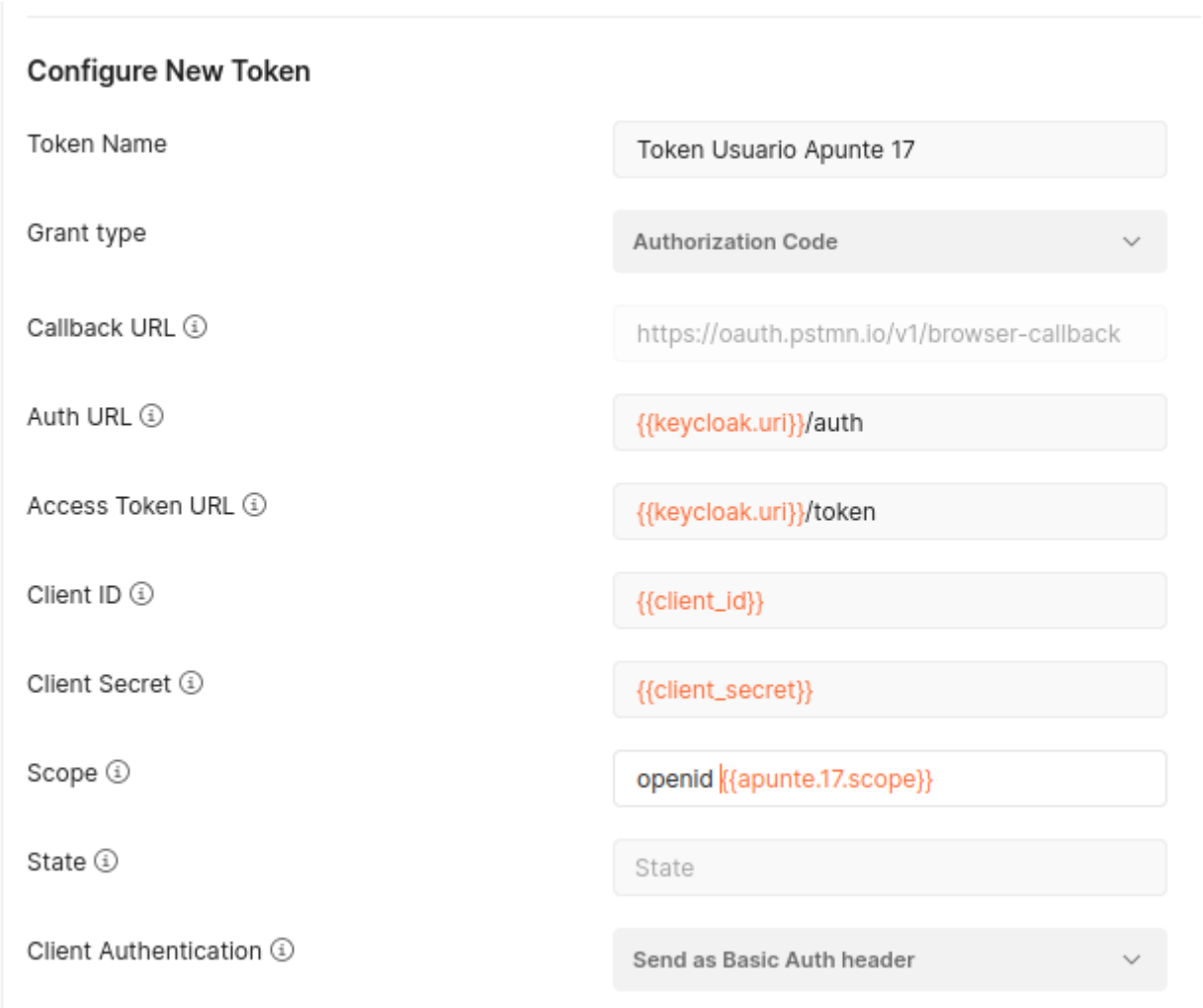
Flujo de Código de Autorización

Este flujo es el recomendado cuando es posible utilizarlo (y es el explicado en el documento de despliegue y configuración de keycloak), en Postman se puede configurar de la siguiente manera:

1. Crear una nueva petición hacia el *Resource Server*. En el ejemplo de este apunte, **GET http://localhost:8080/protegido-usuarios**
2. En la pestaña *Authorization* seleccionar Type: *OAuth 2.0* y Add Authorization Data to: *Request Headers*



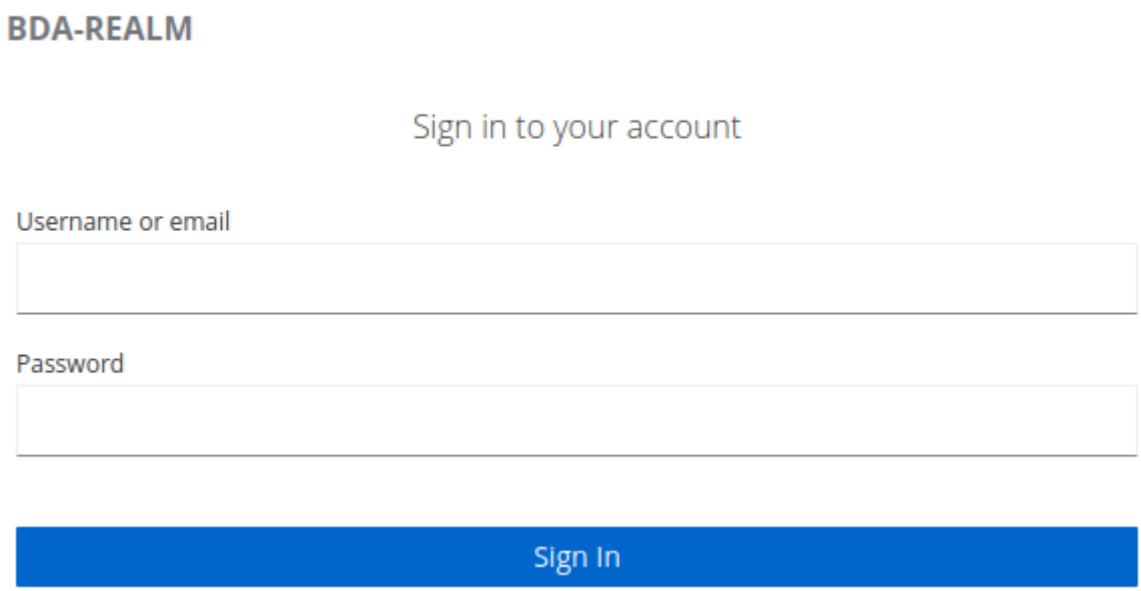
3. En la ventana de la derecha (todavía bajo *Authorization*), ir a la sección *Configure New Token*, configurando los siguientes valores:



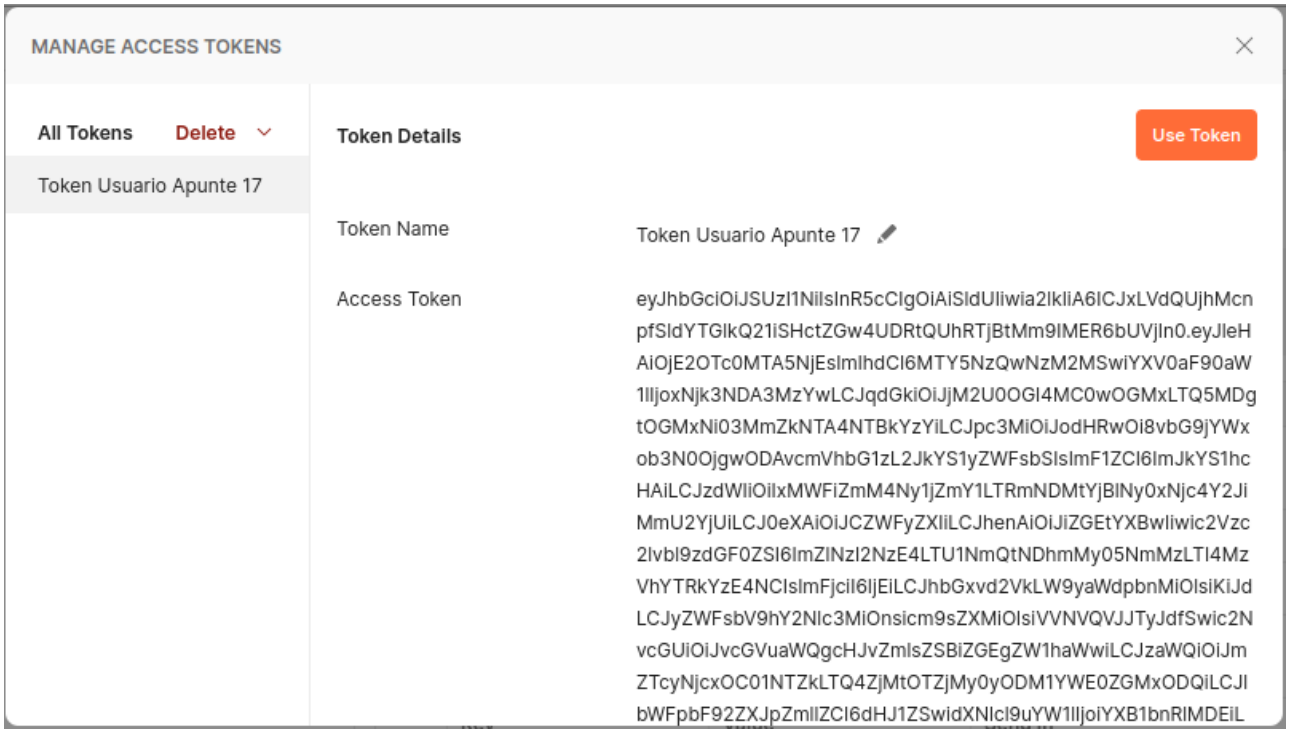
   - {{keycloak.uri}}, {{client_id}}, {{client_secret}}, {{apunte.17.scope}} Corresponden a los valores mencionados previamente. apunte.17.scope, se dejará en blanco

4. Presionar el botón *Get New Access Token*

5. Deberá aparecer una ventana para ingresar las credenciales del usuario (el usuario final que se está autorizando). Colocar aquí las credenciales provistas.



6. Luego de la autenticación, se habrá generado un nuevo token, que puede ser guardado para ser utilizado en las siguientes peticiones. Presionar el botón *Use Token*



7. A partir de este punto, el token estará guardado y se podrá utilizar en cada petición a los endpoints.

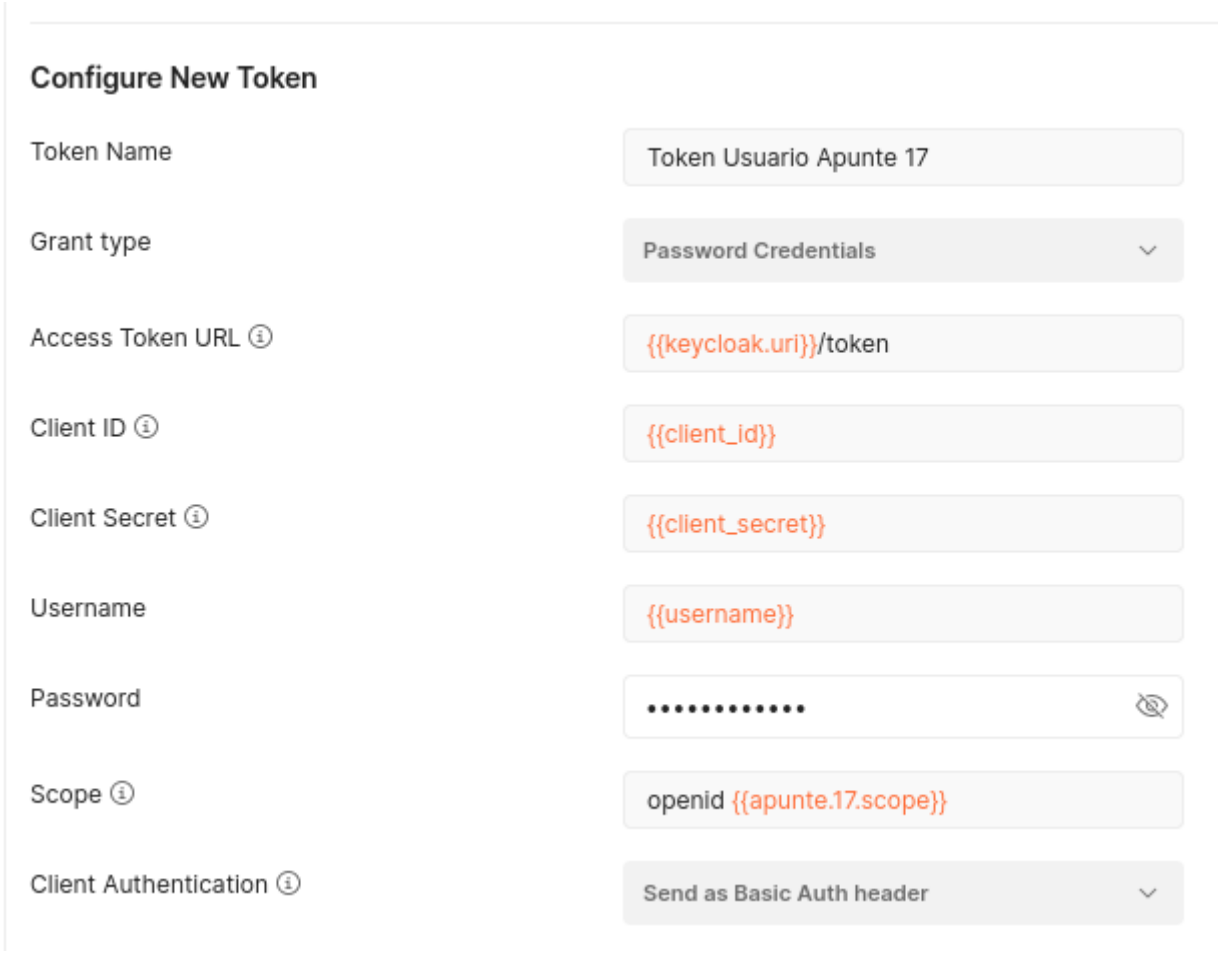
Se puede notar un parámetro adicional "*Callback URL*", esta URL es la explicada en el documento de despliegue y configuración de keycloak, y corresponde al endpoint que intercambia el código de autorización por el token.

Flujo de Resource Owner Password Credentials

Como se estudió en el apunte anterior, no es la mejor de las opciones (porque el usuario debe compartir sus credenciales con el cliente). Sin embargo, para poder realizar las validaciones de la autorización en los

endpoints puede ser una opción más sencilla que la anterior.

La configuración de este flujo es similar a la configuración del Código de Autorización, pero con las siguientes modificaciones:



Grant Type se modifica por *Password Credentials* y debe proveerse el nombre de usuario y el password.